
Chapter 9

SUBPROGRAMS

Chapter 9 Topics

- Introduction
- Fundamentals of Subprograms
- Design Issues for Subprograms
- Local Referencing Environments
- Parameter-Passing Methods
- Parameters That Are Subprograms
- Calling Subprograms Indirectly
- Design Issues for Functions
- Overloaded Subprograms
- Generic Subprograms
- User-Defined Overloaded Operators
- Closures
- Coroutines

Introduction

- Two fundamental abstraction facilities
 - Process abstraction
 - Instead of writing the same steps over and over, we package them into a reusable unit.
 - You created a process abstraction; the details of how the average is computed are hidden behind a name.

```
def compute_average(list):  
    return sum(list) / len(list)
```

- Data abstraction
 - Emphasized in the 1980s
 - Discussed at length in Chapter 11
 - Instead of manipulating raw variables everywhere, we group data and hide its internal representation.

Introduction (process abstraction)

Concept

Subprograms

Parameters

Return values

Recursion

Closures

Coroutines

Examples

functions, procedures, methods

passing data into processes

getting results back

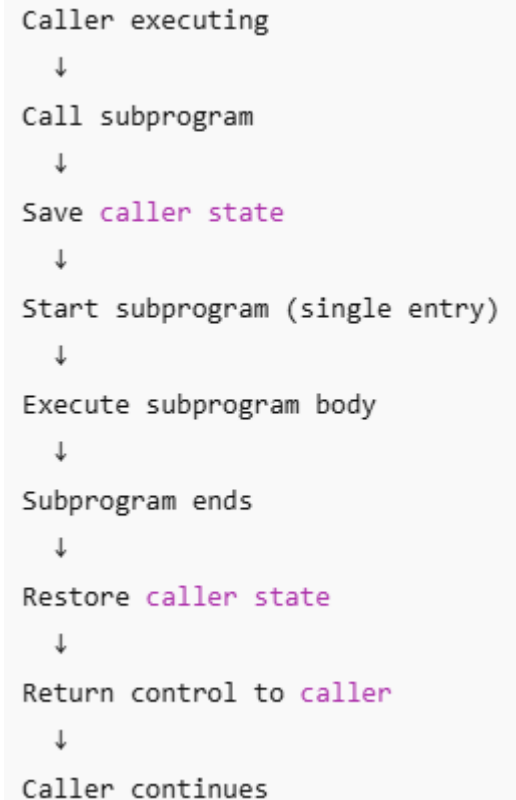
a process defined in terms of itself

processes that remember their environment

cooperative processes

Fundamentals of Subprograms

- Each subprogram has a single-entry point (you cannot jump into the middle of it from outside)
- The calling program is suspended during execution of the called subprogram
- Control always returns to the caller when the called subprogram's execution terminates



Fundamentals Subprograms

```
+-----+
| Code (Text)      | ← compiled functions live here
+-----+
| Global / Static  |
+-----+
| Heap             | ← dynamic memory (new / malloc)
+-----+
| Stack            | ← function calls (stack frames)
+-----+
```

Basic Definitions

- A *subprogram definition* describes the interface to and the actions of the subprogram abstraction
 - In Python, function definitions are executable; in all other languages, they are non-executable
 - In Ruby, function definitions can appear either in or outside of class definitions. If outside, they are methods of Object. They can be called without an object, like a function
 - In Lua, all functions are anonymous
- A *subprogram call* is an explicit request that the subprogram be executed
- A *subprogram header* is the first part of the definition, including the name, the kind of subprogram, and the formal parameters
- The *parameter profile* (aka *signature*) of a subprogram is the number, order, and types of its parameters
- The *protocol* is a subprogram's parameter profile and, if it is a function, its return

Basic Definitions

```
- #include <iostream>
- using namespace std;

- double computeAverage(int total, int count) {
-     return (double) total / count;
- }

- int main() {
-     double result = computeAverage(100, 4);
-     cout << result << endl;
-     return 0;
```

```
• }
```

Concept

Header

Parameter Profile

Signature

Protocol

From the Example

double computeAverage(int total, int count)

(int, int) //number, order and types

computeAverage(int, int)//use for overloading

double computeAverage(int, int), (int int)->double

Basic Definitions

- A *subprogram definition* describes the **interface** to and the **actions** of the subprogram abstraction
- `int max(int a, int b)` -> **The interface** (the “what can I use?” part)
- This tells the outside world:
 - The **name** of the subprogram
 - Its **parameters** (how many, types, order)
 - Its **return value** (if any)
- `int max(int a, int b) {` -> **The actions** (the “how it works” part)
- `if (a > b) return a;`
- `else return b;`
- `}`

Basic Definitions (Python)

In Python, function definitions are **executable**; in all other languages, they are non-executable. **Executable does not mean here computation.**

```
if condition:
    def f(x): // object function created a runtime, return x+1 stored in the object, name f bound object
        return x + 1 //function are run time
else:
    def f(x):
        return x + 2
```

In contrast with **C**

```
if (condition) {
    int f() { return 1}; // type binding, and function
}
```

Step	When it happens
Function type is determined	Compile time
Machine code for f is generated	Compile time
Name f is bound to that code	Link/load time
Function body runs	Only when called

Basic Definitions

- In Python, function definitions are **executable**; in all other languages, they are non-executable
- Here, **executable does not mean perform computations.**
- **In Python**, when execution reaches that line, Python:

```
def f(x):  
    return x + 1
```

Creates a function object. This object contains:

- The code for the body (return x + 1)
- The parameter list
- A reference to the current environment (closure, if needed)
- **Binds the name f to that function object (at run time, dynamic)**
- name → object
- f → <function object>

Basic definitions (functions vs procedures)

In Ruby, function definitions can appear either **in** or **outside** of class definitions. If outside, they are methods of Object. They can be called without an object, like a function

Functions (return) Python

```
def square(x):  
    return x * x
```

Procedures

```
def print_report(name):  
    print("Report for", name)
```

In C++/C

```
int add(int a, int b) {  
    return a + b;  
std::endl;  
}
```

```
void printSum(int a, int b) {  
    std::cout << a + b <<  
}
```

Basic definitions (The case of Ruby)

In Ruby, function definitions can appear either in or outside of class definitions. If outside, they are methods of Object. They can be called without an object, like a function

```
class Dog
  def bark
    "Woof!"
  end
end
```

```
d = Dog.new
```

```
puts d.bark //prints "Woof"
```

```
class Dog
  def bark
    "Woof!"
  end
end
```

```
def greet (name)
  "Hello, #{name}"
end
```

This looks like a **standalone function**, but Ruby secretly does this:

- It defines **greet** as a private method of the class Object
- And since every object in Ruby inherits from Object, the method is available everywhere.

greet("Luis")

```
class Object
  private
  def greet(name)
    "Hello, #{name}"
  end
end
```

Basic definitions (The case of Lua)

In Lua, all functions are anonymous; functions themselves **do not have built-in names**. A “named function” is just a variable that **refers to a function value**. So, the name is **not part of the function**, it’s just a reference.

```
f = function(x)
  return x + 1
end
```

`function(x) return x + 1 end` → creates a function value

`f` → is just a variable pointing to that function. The function itself is nameless

Example

```
f = function(x) return x + 1 end
g = f
f = nil
```

```
print(g(5)) -- still works → 6
```

Basic Definitions (continued)

- Function declarations in C and C++ are often called *prototypes*: `int add(int a, int b);`
- A *subprogram declaration* provides the protocol, but not the body, of the subprogram

`int add(int, int)`

Parameter profile → (int, int)

Return type → int

- A *formal parameter* is a dummy variable listed in the subprogram header and used in the subprogram

`int add(int a, int b)`

- An *actual parameter* represents a value or address used in the subprogram call statement

`int result = add(5, 3);`

Basic definitions

```
#include <iostream>
```

```
int multiply(int x, int y); // Prototype (declaration)
```

```
int main() {  
    int result = multiply(4, 6); // 4 and 6 are ACTUAL parameters  
    std::cout << result << std::endl;  
    return 0;  
}
```

```
// Function definition  
int multiply(int x, int y) { // x and y are FORMAL parameters  
    return x * y;  
}
```

Actual/Formal Parameter Correspondence

- **Positional**

- The binding of actual parameters to formal parameters is by position: the first actual parameter is bound to the first formal parameter, and so forth
- Safe and effective

```
def move(x, y, speed): -> move(10, 20, 5)
```

```
def schedule_meeting(title, date, *, time, location="TBD"): -> schedule_meeting("Sync", "Feb 5",  
time="10:00")
```

- **Keyword**

- The name of the formal parameter to which an actual parameter is to be bound is specified with the actual parameter, after *** should be keyword
- *Advantage*: Parameters can appear in any order, thereby avoiding parameter correspondence errors
- *Disadvantage*: User must know the formal parameter's names

```
def send_email(*, to, subject, body): %all parameters after * by keyword
```

```
send_email(subject="Meeting", to="ana@mail.com", body="See you at 3 PM")
```

Formal Parameter Default Values

- In certain languages (e.g., C++, Python, Ruby, PHP), formal parameters can have default values (if no actual parameter is passed)
 - In C++, default parameters must appear last because parameters are positionally associated (no keyword parameters)

`void greet(string name, string message = "Hello", int times = 1);`

Once a parameter has a default value, **all parameters to its right must also have defaults.**

- Variable numbers of parameters
 - C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **params**

– In Ruby, the actual parameters are sent as elements of a hash literal and the corresponding formal parameter is

Formal Parameter Default Values

Variable numbers of parameters

- C# methods can accept a variable number of parameters as long as they are of the same type—the corresponding formal parameter is an array preceded by **params**

```
void PrintNumbers(params int[] numbers)
{
    foreach (int n in numbers)
    {
        Console.WriteLine(n);
    }
}
PrintNumbers(1);
PrintNumbers(1, 2, 3, 4);
PrintNumbers();    // also valid
```

Formal Parameter Default Values

Ruby uses `*` in the formal parameter list.

`*numbers` collects positional arguments into an array

`**options` collects keyword arguments into a hash

```
def print_numbers(*numbers)
  numbers.each { |n| puts n }# { |n|, ...} the block
end
```

```
print_numbers(1) #makes numbers =[1]
```

```
print_numbers(1, 2, 3, 4) #makes numbers =[1, 2, 3, 4]
```

```
print_numbers()
```

keyword-style variable parameters (Hash), gather named arguments into a hash using `**`.

```
def create_user(**info)
  puts info[:name]
  puts info[:age]
End
```

Call: `create_user(name: "Luis", age: 35, admin: true)`

```
info = { name: "Luis", age: 35, admin: true }
```

Variable Numbers of Parameters

(co

In Python, the actual is a list of values, and the corresponding formal parameter is a name with an asterisk

```
def print_numbers(*numbers):  
    for n in numbers:  
        print(n)
```

*numbers means: collect extra positional arguments Inside the function, numbers is a tuple

```
print_numbers(1)  
print_numbers(1, 2, 3, 4)          numbers = (1, 2, 3, 4)  
print_numbers()
```

variable number of keyword arguments → **kwargs

```
def create_user(**info):  
    print(info["name"])  
    print(info["age"])
```

```
create_user(name="Luis", age=35)
```

Variable Numbers of Parameters

```
def demo(a, b, *args, **kwargs):  
    print("a:", a)  
    print("b:", b)  
    print("extra positional:", args)  
    print("extra keywords:", kwargs)
```

call: demo(1, 2, 3, 4, x=10, y=20)

Inside

a = 1

b = 2

args = (3, 4) %**extra positional arguments** into a tuple

kwargs = {"x": 10, "y": 20} %**keyword arguments** into a dictionary

Language

C#

Ruby

Python

Variable positional

params int[] nums

*nums (Array)

*args (Tuple)

Variable keyword

□ Not built-in

**info (Hash)

**kwargs (Dictionary)

Procedures and Functions

- There are two categories of subprograms
 - *Procedures* are a collection of statements that define parameterized computations

```
void printSum(int a, int b) {  
    //They are expected to produce no side effects  
    printf("%d\n", a + b);  
}
```

- *Functions* structurally resemble procedures but are semantically modeled on mathematical functions, $f(x)=x^2$

Design Issues for Subprograms

- Are local variables static or dynamic?
- Can subprogram definitions appear in other subprogram definitions?
- What parameter passing methods are provided?
- Are parameter types checked?
- If subprograms can be passed as parameters and subprograms can be nested, what is the referencing environment of a passed subprogram?
- Are functional side effects allowed?
- What types of values can be returned from functions?
- How many values can be returned from functions?
- Can subprograms be overloaded?
- Can subprogram be generic?
- If the language allows nested subprograms, are closures supported?

Local Referencing Environments

- Local variables can be stack-dynamic
 - Advantages
 - Support for recursion
 - Storage for locals is shared among some subprograms
 - Disadvantages
 - Allocation/de-allocation, initialization time
 - Indirect addressing
 - Subprograms cannot be history sensitive
- Local variables can be static
 - Advantages and disadvantages are the opposite of those for stack-dynamic local variables

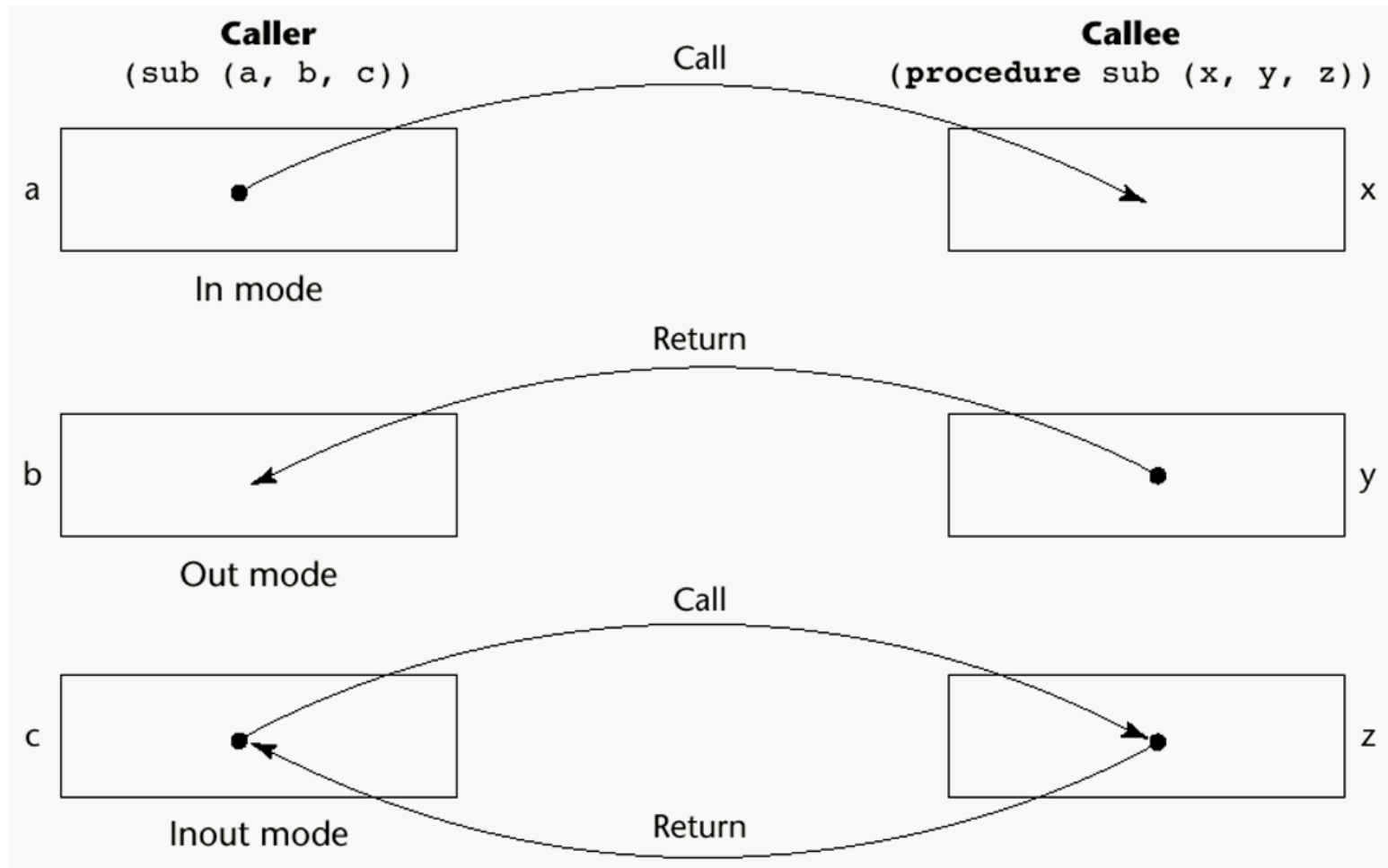
Local Referencing Environments: Examples

- In most contemporary languages, locals are stack dynamic
- In C-based languages, locals are by default stack dynamic, but can be declared `static`
- The methods of C++, Java, Python, and C# only have stack dynamic locals

Semantic Models of Parameter Passing

- In mode
- Out mode
- Inout mode

Models of Parameter Passing



Conceptual Models of Transfer

- Physically move a value
- Move an access path to a value

Pass-by-Value (In Mode)

- The value of the actual parameter is used to initialize the corresponding formal parameter
 - Normally implemented by copying
 - Can be implemented by transmitting an access path but not recommended (enforcing write protection is not easy)
 - *Disadvantages* (if by physical move): additional storage is required (stored twice) and the actual move can be costly (for large parameters)
 - *Disadvantages* (if by access path method): must write-protect in the called subprogram and accesses cost more (indirect addressing)

Pass-by-Result (Out Mode)

- When a parameter is passed by result, no value is transmitted to the subprogram; the corresponding formal parameter acts as a local variable; its value is transmitted to caller's actual parameter when control is returned to the caller, by physical move
 - Require extra storage location and copy operation
- Potential problems:
 - `sub(p1, p1);` whichever formal parameter is copied back will represent the current value of `p1`
 - `sub(list[sub], sub);` Compute address of `list[sub]` at the beginning of the subprogram or end?

Pass-by-Value-Result (inout Mode)

- A combination of pass-by-value and pass-by-result
- Sometimes called pass-by-copy
- Formal parameters have local storage
- Disadvantages:
 - Those of pass-by-result
 - Those of pass-by-value

Pass-by-Reference (Inout Mode)

- Pass an access path
 - Also called pass-by-sharing
 - Advantage: Passing process is efficient (no copying and no duplicated storage)
 - Disadvantages
 - Slower accesses (compared to pass-by-value) to formal parameters
 - Potentials for unwanted side effects (collisions)
 - Unwanted aliases (access broadened)
- ```
fun(total, total); fun(list[i], list[j]); fun(list[i], i);
```

# Pass-by-Reference (continued)

---

- Another issue:

Can the passed reference be changed in the called subprogram?

- In C, it is possible
- But in some other languages, such as Pascal and C++, formal parameters that are addresses are implicitly dereferenced, which prevents such changes

# Pass-by-Name (Inout Mode)

---

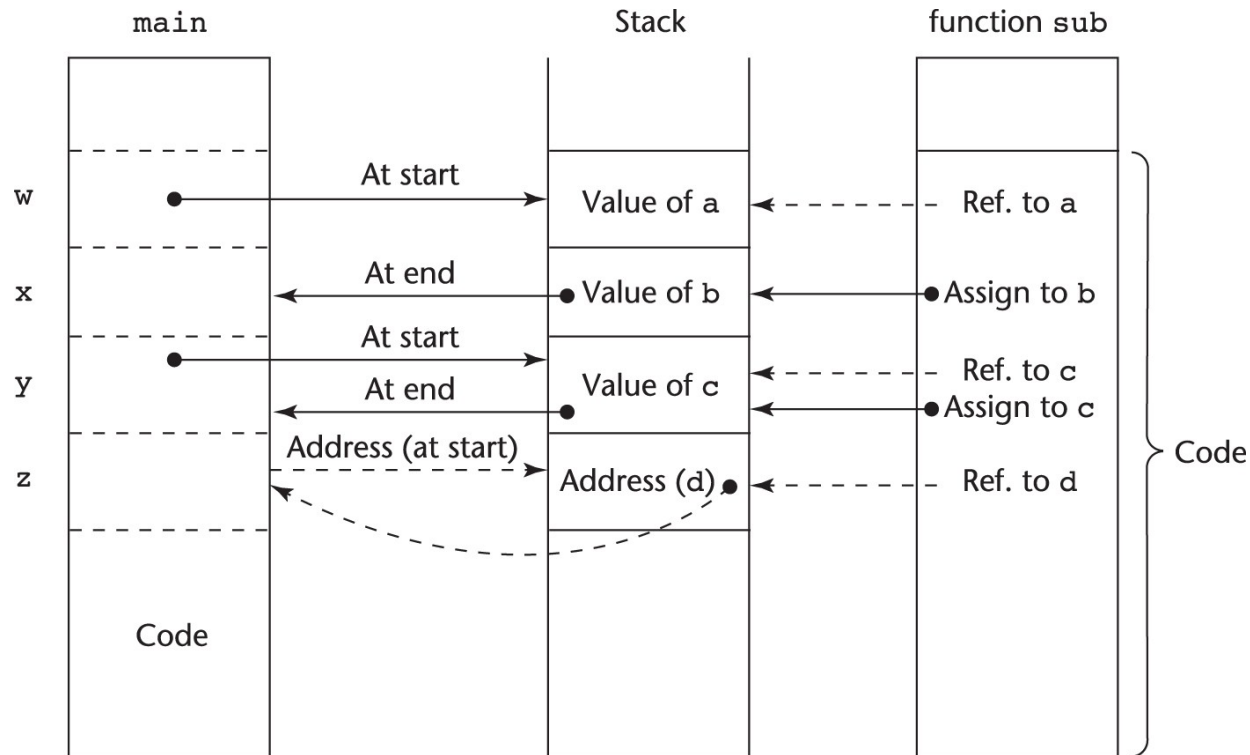
- By textual substitution
- Formals are bound to an access method at the time of the call, but actual binding to a value or address takes place at the time of a reference or assignment
- Allows flexibility in late binding
- Implementation requires that the referencing environment of the caller is passed with the parameter, so the actual parameter address can be calculated

# Implementing Parameter-Passing Methods

---

- In most languages parameter communication takes place thru the run-time stack
- Pass-by-reference are the simplest to implement; only an address is placed in the stack

# Implementing Parameter-Passing Methods



Function header: **void sub(int a, int b, int c, int d)**

Function call in main: **sub(w, x, y, z)**

(pass **w** by value, **x** by result, **y** by value-result, **z** by reference)

# Parameter Passing Methods of Major Languages

---

- C
  - Pass-by-value
  - Pass-by-reference is achieved by using pointers as parameters
- C++
  - A special pointer type called reference type for pass-by-reference
- Java
  - All non-object parameters are passed are passed by value  
So, no method can change any of these parameters
  - Object parameters are passed by reference

# Parameter Passing Methods of Major Languages (continued)

---

- Fortran 95+
  - Parameters can be declared to be in, out, or inout mode
- C#
  - Default method: pass-by-value
    - Pass-by-reference is specified by preceding both a formal parameter and its actual parameter with `ref`
- PHP: very similar to C#, except that either the actual or the formal parameter can specify `ref`
- Swift: default passing method is by value, but pass-by-reference can be specified by preceding the formal with **`inout`**
- Perl: all actual parameters are implicitly placed in a predefined array named `@_`
- Python and Ruby use pass-by-assignment (all data values are objects); the actual is assigned to the formal

# Type Checking Parameters

---

- Considered very important for reliability
- FORTRAN 77 and original C: none
- Pascal and Java: it is always required
- ANSI C and C++: choice is made by the user
  - Prototypes
- Relatively new languages Perl, JavaScript, and PHP do not require type checking
- In Python and Ruby, variables do not have types (objects do), so parameter type checking is not possible

# Multidimensional Arrays as Parameters

---

- If a multidimensional array is passed to a subprogram and the subprogram is separately compiled, the compiler needs to know the declared size of that array to build the storage mapping function

# Multidimensional Arrays as Parameters: C and C++

---

- Programmer is required to include the declared sizes of all but the first subscript in the actual parameter
- Disallows writing flexible subprograms
- Solution: pass a pointer to the array and the sizes of the dimensions as other parameters; the user must include the storage mapping function in terms of the size parameters

# Multidimensional Arrays as Parameters: Java and C#

---

- Similar to Ada
- Arrays are objects; they are all single-dimensioned, but the elements can be arrays
- Each array inherits a named constant (`length` in Java, `Length` in C#) that is set to the length of the array when the array object is created

# Design Considerations for Parameter Passing

---

- Two important considerations
  - Efficiency
  - One-way or two-way data transfer
- But the above considerations are in conflict
  - Good programming suggest limited access to variables, which means one-way whenever possible
  - But pass-by-reference is more efficient to pass structures of significant size

# Parameters that are Subprogram Names

---

- It is sometimes convenient to pass subprogram names as parameters
- Issues:
  1. Are parameter types checked?
  2. What is the correct referencing environment for a subprogram that was sent as a parameter?

# Parameters that are Subprogram Names: Referencing Environment

- *Shallow binding*: The environment of the call statement that enacts the passed subprogram
  - Most natural for dynamic-scoped languages
- *Deep binding*: The environment of the definition of the passed subprogram
  - Most natural for static-scoped languages
- *Ad hoc binding*: The environment of the call statement that passed the subprogram

# Calling Subprograms Indirectly

---

- Usually when there are several possible subprograms to be called and the correct one on a particular run of the program is not known until execution (e.g., event handling and GUIs)
- In C and C++, such calls are made through function pointers

# Calling Subprograms Indirectly (continued)

---

- In C#, method pointers are implemented as objects called *delegates*
  - A delegate declaration:  
**public delegate int** Change(**int** x);
    - This delegate type, named Change, can be instantiated with any method that takes an **int** parameter and returns an **int** value
  - A method: **static int** fun1(**int** x) { ... }
  - Instantiate: Change chgfun1 = **new** Change(fun1);
  - Can be called with: chgfun1(12);
  - A delegate can store more than one address, which is called a *multicast delegate*

# Design Issues for Functions

---

- Are side effects allowed?
  - Parameters should always be in-mode to reduce side effect (like Ada)
- What types of return values are allowed?
  - Most imperative languages restrict the return types
  - C allows any type except arrays and functions
  - C++ is like C but also allows user-defined types
  - Java and C# methods can return any type (but because methods are not types, they cannot be returned)
  - Python and Ruby treat methods as first-class objects, so they can be returned, as well as any other class

# Overloaded Subprograms

---

- An *overloaded subprogram* is one that has the same name as another subprogram in the same referencing environment
  - Every version of an overloaded subprogram has a unique protocol
- C++, Java, C#, and Ada include predefined overloaded subprograms
- In Ada, the return type of an overloaded function can be used to disambiguate calls (thus two overloaded functions can have the same parameters)
- Ada, Java, C++, and C# allow users to write multiple versions of subprograms with the same name

# Generic Subprograms

---

- A *generic* or *polymorphic subprogram* takes parameters of different types on different activations
- Overloaded subprograms provide *ad hoc polymorphism*
- *Subtype polymorphism* means that a variable of type T can access any object of type T or any type derived from T (OOP languages)
- A subprogram that takes a generic parameter that is used in a type expression that describes the type of the parameters of the subprogram provides *parametric polymorphism*
  - A cheap compile-time substitute for dynamic binding

# Generic Subprograms (continued)

---

- C++
  - Versions of a generic subprogram are created implicitly when the subprogram is named in a call or when its address is taken with the & operator
  - Generic subprograms are preceded by a **template** clause that lists the generic variables, which can be type names or class names

```
template <class Type>
 Type max(Type first, Type second) {
 return first > second ? first : second;
 }
```

# Generic Subprograms (continued)

---

- Java 5.0
  - Differences between generics in Java 5.0 and those of C++:
    1. Generic parameters in Java 5.0 must be classes
    2. Java 5.0 generic methods are instantiated just once as truly generic methods
    3. Restrictions can be specified on the range of classes that can be passed to the generic method as generic parameters
    4. Wildcard types of generic parameters

# Generic Subprograms (continued)

---

- Java 5.0 (continued)

```
public static <T> T doIt(T[] list) { ... }
```

- The parameter is an array of generic elements (T is the name of the type)

- A call:

```
doIt<String>(myList);
```

Generic parameters can have bounds:

```
public static <T extends Comparable> T
doIt(T[] list) { ... }
```

The generic type must be of a class that implements the Comparable interface

# Generic Subprograms (continued)

---

- Java 5.0 (continued)

- Wildcard types

`Collection<?>` is a wildcard type for collection classes

```
void printCollection(Collection<?> c) {
 for (Object e: c) {
 System.out.println(e);
 }
}
```

- Works for any collection class

# Generic Subprograms (continued)

---

- C# 2005
  - Supports generic methods that are similar to those of Java 5.0
  - One difference: actual type parameters in a call can be omitted if the compiler can infer the unspecified type
    - Another – C# 2005 does not support wildcards

# Generic Subprograms (continued)

---

- F#

- Infers a generic type if it cannot determine the type of a parameter or the return type of a function – *automatic generalization*
- Such types are denoted with an apostrophe and a single letter, e.g., 'a
- Functions can be defined to have generic parameters

```
let printPair (x: 'a) (y: 'a) =
 printfn "%A %A" x y
```

- %A is a format code for any type
- These parameters are not type constrained

# Generic Subprograms (continued)

---

- F# (continued)
  - If the parameters of a function are used with arithmetic operators, they are type constrained, even if the parameters are specified to be generic
  - Because of type inferencing and the lack of type coercions, F# generic functions are far less useful than those of C++, Java 5.0+, and C# 2005+

# User-Defined Overloaded Operators

---

- Operators can be overloaded in Ada, C++, Python, and Ruby
- A Python example

```
def __add__(self, second) :
 return Complex(self.real + second.real,
 self.imag + second.imag)
```

Use: To compute  $x + y$ , `x.__add__(y)`

# Closures

---

- A *closure* is a subprogram and the referencing environment where it was defined
  - The referencing environment is needed if the subprogram can be called from any arbitrary place in the program
  - A static-scoped language that does not permit nested subprograms doesn't need closures
  - Closures are only needed if a subprogram can access variables in nesting scopes and it can be called from anywhere
  - To support closures, an implementation may need to provide unlimited extent to some variables (because a subprogram may access a nonlocal variable that is normally no longer alive)

# Closures (continued)

---

- A JavaScript closure:

```
function makeAdder(x) {
 return function(y) {return x + y;}
}
```

...

```
var add10 = makeAdder(10);
```

```
var add5 = makeAdder(5);
```

```
document.write("add 10 to 20: " + add10(20) +
 "
");
```

```
document.write("add 5 to 20: " + add5(20) +
 "
");
```

- The closure is the anonymous function returned by makeAdder

# Closures (continued)

---

- C#

- We can write the same closure in C# using a nested anonymous delegate
- `Func<int, int>` (the return type) specifies a delegate that takes an `int` as a parameter and returns an `int`

```
static Func<int, int> makeAdder(int x) {
 return delegate(int y) {return x + y;};
}
```

```
...
```

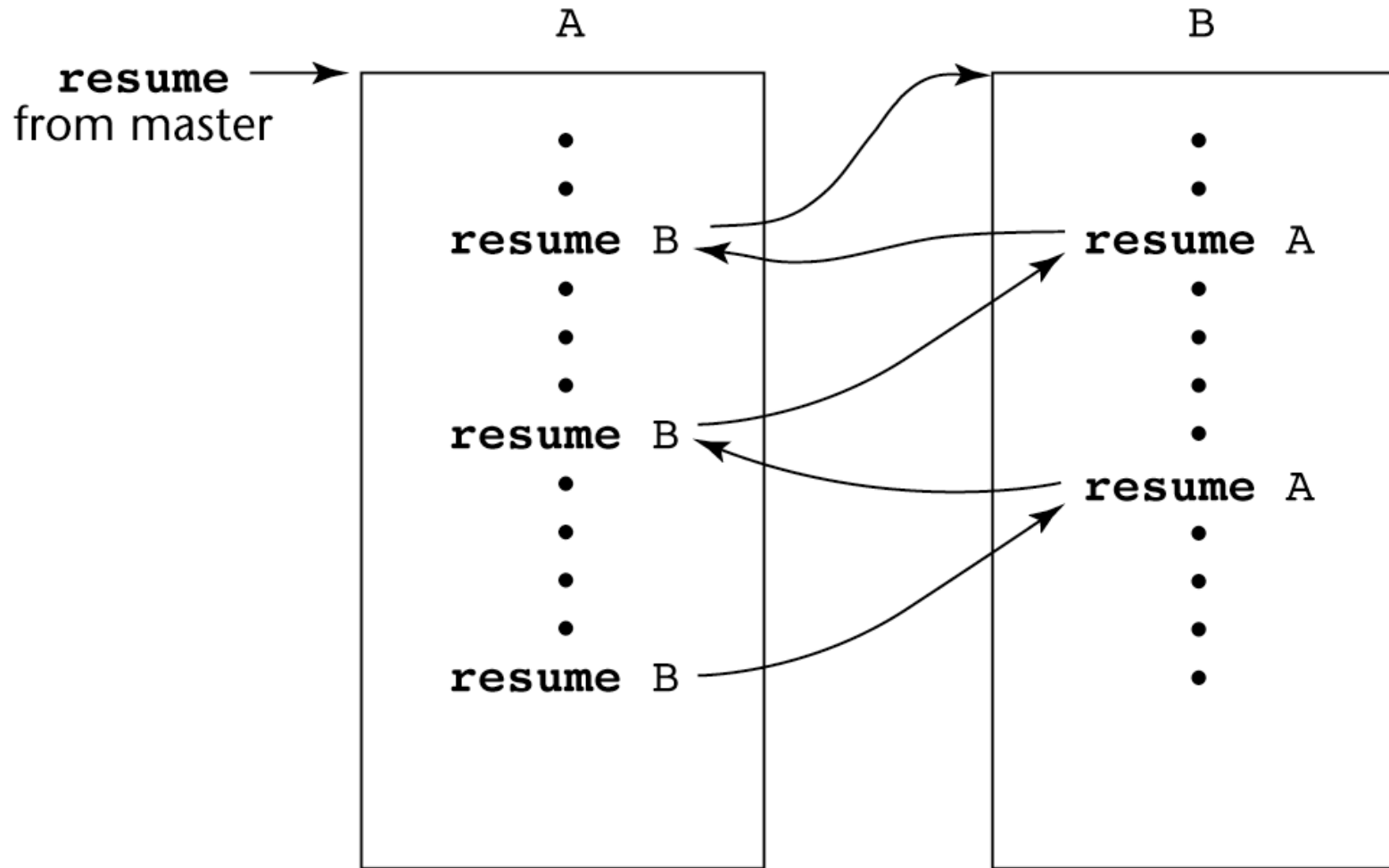
```
Func<int, int> Add10 = makeAdder(10);
Func<int, int> Add5 = makeAdder(5);
Console.WriteLine("Add 10 to 20: {0}",
 Add10(20));
Console.WriteLine("Add 5 to 20: {0}", Add5(20));
```

# Coroutines

---

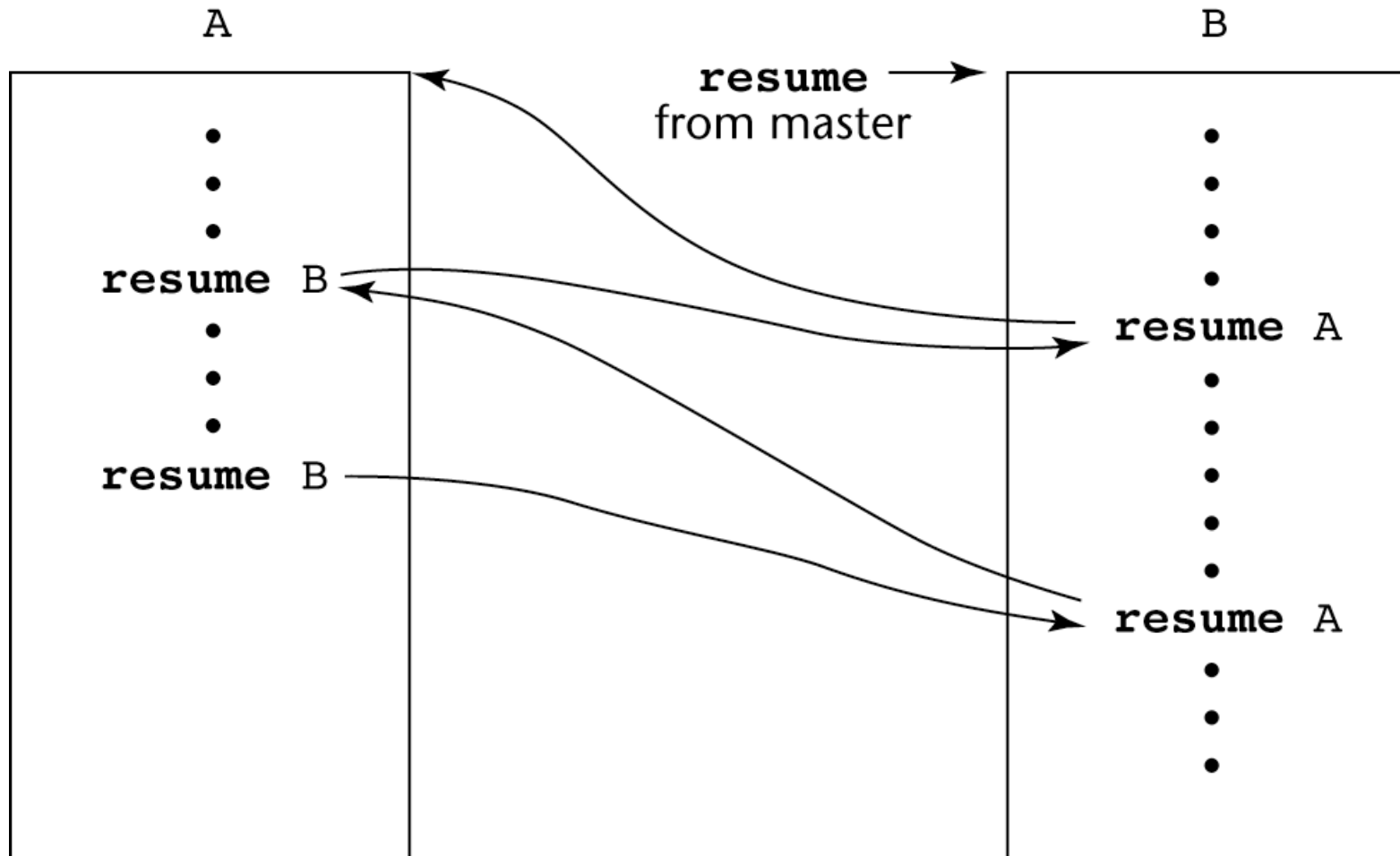
- A *coroutine* is a subprogram that has multiple entries and controls them itself – supported directly in Lua
- Also called *symmetric control*: caller and called coroutines are on a more equal basis
- A coroutine call is named a *resume*
- The first resume of a coroutine is to its beginning, but subsequent calls enter at the point just after the last executed statement in the coroutine
- Coroutines repeatedly resume each other, possibly forever
- Coroutines provide *quasi-concurrent execution* of program units (the coroutines); their execution is interleaved, but not overlapped

# Coroutines Illustrated: Possible Execution Controls



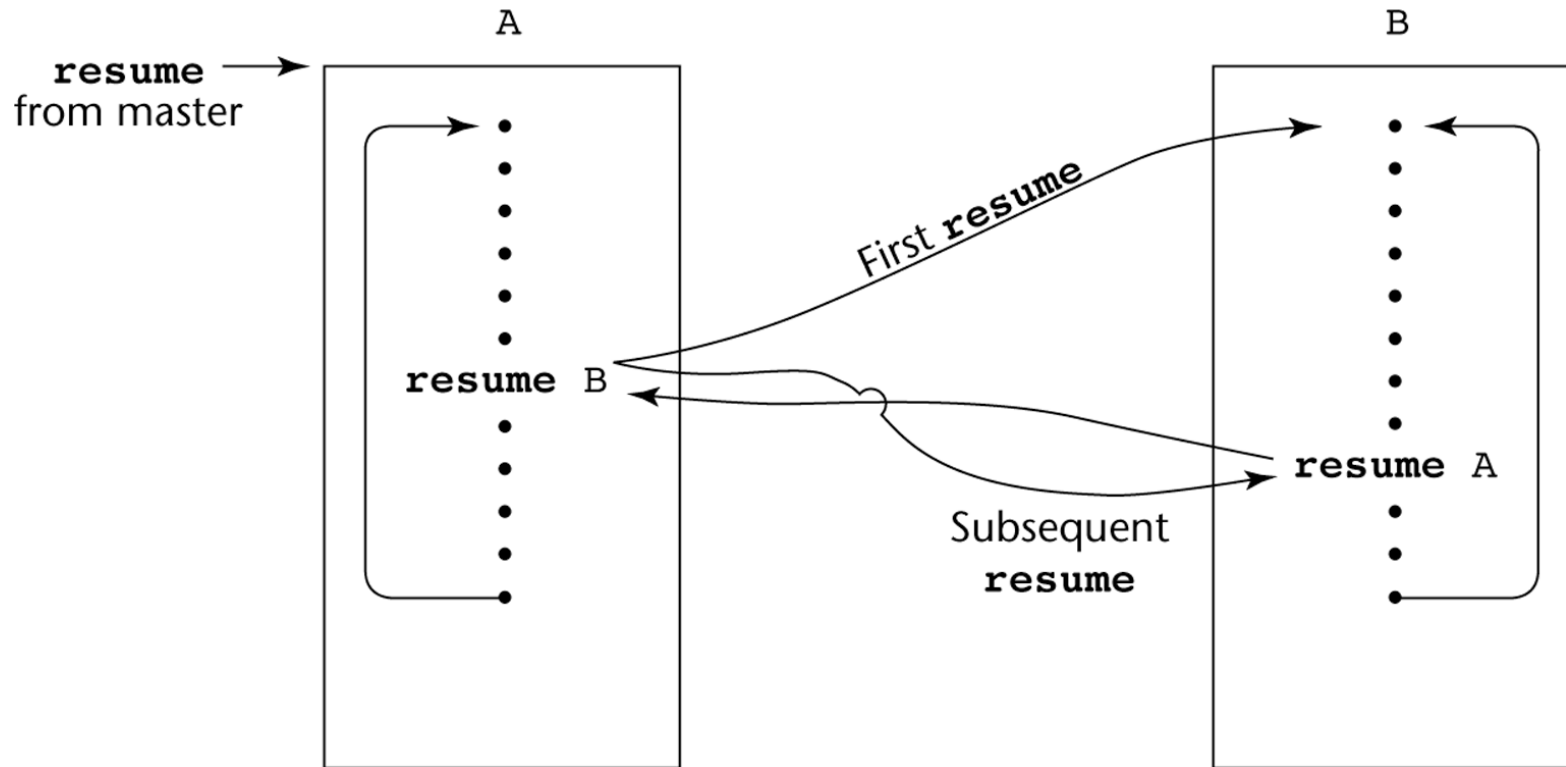
(a)

# Coroutines Illustrated: Possible Execution Controls



(b)

# Coroutines Illustrated: Possible Execution Controls with Loops



# Summary

---

- A subprogram definition describes the actions represented by the subprogram
- Subprograms can be either functions or procedures
- Local variables in subprograms can be stack-dynamic or static
- Three models of parameter passing: in mode, out mode, and inout mode
- Some languages allow operator overloading
- Subprograms can be generic
- A closure is a subprogram and its ref. environment
- A coroutine is a special subprogram with multiple entries